

ArtifactSync: Automated Repository Synchronization through Hierarchical Change Impact Analysis

Ebube Alor

ebubechukwu.alor@mail.concordia.ca
Concordia University
Montréal, Quebec, Canada

SayedHassan Khatoonabadi

sayedhassan.khatoonabadi@concordia.ca
Concordia University
Montréal, Quebec, Canada

João Pedro de Souza Olivo Tardivo

joao.tardivo.15@estudante.unespar.edu.br
Universidade Estadual do Paraná
Apucarana, Paraná, Brazil

Emad Shihab

emad.shihab@concordia.ca
Concordia University
Montréal, Quebec, Canada

Abstract

When developers change source code, related artifacts like documentation and tests often become outdated, creating inconsistencies that are difficult to fix manually. We introduce *ArtifactSync*, a tool that leverages Large Language Models (LLMs) for automated cross-artifact consistency maintenance. LLMs are uniquely suited for this task because they can understand semantic relationships across diverse artifact types, such as code, documentation, and tests that traditional static analysis cannot capture. *ArtifactSync* performs change impact analysis using a hierarchical approach that starts by scanning file names and progressively “zooms in” on uncertain files, requesting more detailed context only when necessary. This targeted change impact analysis makes the evaluation of large repositories both feasible and cost-effective. *ArtifactSync* is implemented as both a command-line tool and an interactive VS Code extension. Within the editor, it presents each detected inconsistency with a clear explanation and an actionable fix that developers can apply directly. Our preliminary evaluation across two open-source repositories shows that *ArtifactSync* achieved up to 100% accuracy in both identifying impacted artifacts and generating corresponding fixes. A demonstration is available at https://youtu.be/AfLe_I_OnA0 and the tool at <https://github.com/Alor-e/artifact-sync>.

CCS Concepts

• **Software and its engineering** → **Software maintenance tools**; *Software maintenance*.

Keywords

Change Impact Analysis, Software Artifact Synchronization, Hierarchical Context Retrieval, Automatic Fix Suggestion

ACM Reference Format:

Ebube Alor, João Pedro de Souza Olivo Tardivo, SayedHassan Khatoonabadi, and Emad Shihab. 2026. *ArtifactSync: Automated Repository Synchronization through Hierarchical Change Impact Analysis*. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26)*,



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICSE-Companion '26, Rio de Janeiro, Brazil*

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2296-7/2026/04
<https://doi.org/10.1145/3774748.3787617>

April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 4 pages.
<https://doi.org/10.1145/3774748.3787617>

1 Introduction

Modern software projects consist of numerous interconnected artifacts. Beyond the source code, these include documentation, unit tests, build scripts, and configuration files, all of which are essential for the development, maintenance, and comprehension of the software [16]. A critical challenge in the software lifecycle is maintaining consistency and synchronization among these diverse artifacts. When a developer introduces a new feature or refactors existing code, corresponding changes are often required in other parts of the repository. Failure to update all related artifacts in lockstep leads to inconsistencies, which manifest as outdated documentation, failing or missing tests, and broken builds [11, 15, 16]. This “artifact drift” contributes significantly to technical debt, complicates onboarding for new developers, and increases the overall cost of software maintenance [1, 11, 15, 16].

Currently, developers rely on a combination of manual diligence, code reviews, static analysis tools, and Continuous Integration (CI) pipelines to manage these dependencies. However, these approaches have limitations. Manual checks place a heavy cognitive load on developers, who must remember every implication of their changes [16]. Static analysis tools can detect syntactic relationships but struggle with semantic dependencies across different artifact types. While CI pipelines can detect some issues, such as failing tests, they cannot identify what is missing, such as the absence of a test for a new public method or the need for a documentation update [4]. Recent advances in Large Language Models (LLMs) have shown remarkable capabilities in code understanding and reasoning [3]. LLMs can understand semantic relationships across different artifact types that traditional approaches cannot capture. However, leveraging LLMs to perform comprehensive change impact analysis across repositories, identifying and rectifying artifact inconsistencies introduced by commits, remains a largely unexplored and challenging problem.

To address this challenge, we introduce *ArtifactSync*, a novel tool that automates the detection (via change impact analysis) and resolution of inconsistencies among software artifacts. *ArtifactSync* is available as both a command-line tool and a VS Code extension, designed to integrate seamlessly into a developer’s workflow by

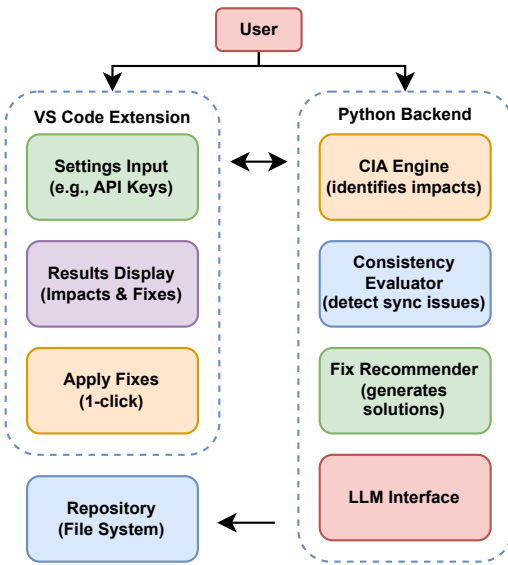


Figure 1: System architecture of ArtifactSync showing the interaction between the frontend and backend.

analyzing the impact of recent code changes across the entire repository. The tool operates by first examining the latest commit to understand the changes made to the source code. It then intelligently identifies related artifacts—such as code files, documentation files, test suites, or configuration scripts—that may be out-of-sync. For each detected inconsistency, ArtifactSync provides a clear, natural language explanation of the inconsistency, suggests a concrete code or text-based fix, and offers the developer the option to automatically apply the correction.

To evaluate ArtifactSync, we defined 20 common inconsistency scenarios that practitioners frequently encounter, such as adding a feature without updating documentation or writing code without corresponding tests, across two open-source repositories. Our preliminary evaluation shows ArtifactSync is highly effective, identifying inconsistencies with up to 100% accuracy in common scenarios and generating appropriate fixes for the majority of them. To facilitate reproducibility and encourage further research, the source code for the command-line backend is publicly available¹ and the repository for the VS Code extension includes a pre-built installer². The remainder of this paper is organized as follows: Section 2 details the design and architecture of ArtifactSync. Section 3 describes the core technologies used for its implementation. Section 4 presents our preliminary evaluation methodology and results. Section 5 reviews related work, and Section 6 concludes the paper and discusses future work.

2 Tool Design

ArtifactSync is designed as a two-component system that performs automated change impact analysis and consistency maintenance across software repositories. As shown in Figure 1, the architecture separates user interaction (VS Code extension) from

¹<https://github.com/Alor-e/artifact-sync-backend>

²<https://github.com/Alor-e/artifact-sync>

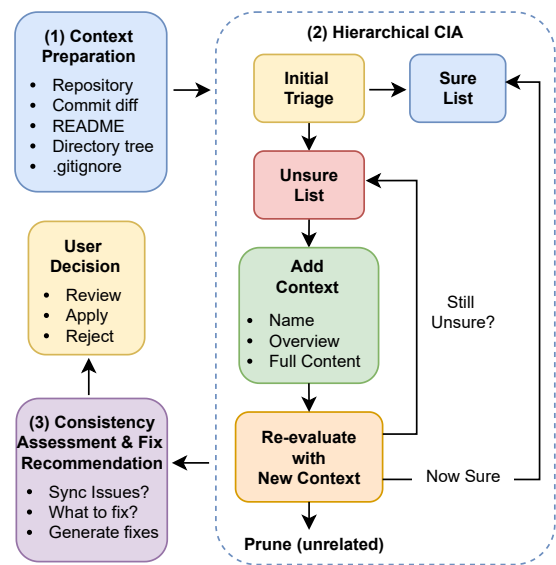


Figure 2: Hierarchical change impact analysis workflow of ArtifactSync showing the main phases.

the analysis engine (Python backend), enabling both command-line and IDE-integrated workflows. This separation allows developers to choose their preferred interaction mode while maintaining consistent analysis capabilities.

The core innovation of ArtifactSync lies in its hierarchical approach to change impact analysis. Rather than processing an entire repository at once (which would be computationally expensive and exceed LLM context limits) the tool strategically reduces the search space by progressively focusing on relevant artifacts. This approach makes repository-wide analysis both feasible and more cost-effective, enabling the tool to handle large-scale projects that would otherwise be impractical.

2.1 Design Rationale

ArtifactSync focuses on analyzing commits rather than uncommitted changes for several key reasons. Commits represent atomic units of change with clear boundaries and developer intent. They provide a stable reference point for analysis and ensure reproducibility. Additionally, commit messages often contain valuable context about the nature and purpose of changes, which aids in understanding their potential impact on related artifacts.

The tool employs two complementary strategies to achieve scalability: (1) **Hierarchical directory traversal**: Starting with a broad view of the repository structure and adaptively drilling down into specific directories only when necessary. (2) **Progressive content resolution**: Examining files at increasing levels of detail (name → overview → full content) based on uncertainty about their relevance.

These strategies work together to minimize the amount of context processed while maximizing the accuracy of impact detection. By requesting additional detail only for uncertain cases, the system avoids the cost and complexity of analyzing every file in detail.

We use the LLM to guide the traversal: it decides what to open next, what to skip, and how much to read, keeping repo-wide CIA scalable.

2.2 Analysis Workflow

Figure 2 illustrates the complete workflow of ArtifactSync, which consists of three main phases:

Phase 1: Context Preparation. The analysis begins when a user provides a repository and specifies a target commit to analyze. The system first applies .gitignore rules to exclude irrelevant files, reducing the search space before analysis begins. It then gathers essential context including the repository’s file tree structure, the README content for project understanding, and the commit diff showing what changed. This lightweight initial context serves as the foundation for subsequent analysis while avoiding the overhead of loading unnecessary file contents.

Phase 2: Hierarchical Change Impact Analysis (CIA). This phase represents the core of ArtifactSync’s approach. As detailed in Figure 2, the system performs an iterative process to identify which artifacts are impacted by the commit:

The process begins with an initial triage that classifies all repository items into two lists based on their apparent relationship to the commit: **Sure list:** Artifacts that appear to be impacted by the commit based on available information. **Unsure list:** Artifacts where the impact is plausible but cannot be determined without additional context.

For items in the Unsure list, the system enters an iterative refinement loop. In each iteration, the LLM re-evaluates uncertain artifacts with progressively more detailed context. When the LLM requires more information to make a determination, it can request: **File overview:** Function signatures, class definitions, and structural information; **Full file contents:** Complete source code when structural information is insufficient; **Subdirectory analysis:** Recursive examination of nested directories when needed.

Based on the additional context, artifacts are either moved to the Sure list, pruned as unrelated, or kept for further refinement. This “just-in-time” context provisioning ensures that expensive operations, like loading entire files, are performed only when necessary. The loop continues until the Unsure list is empty or a predefined iteration limit is reached, at which point remaining items are pruned to prevent infinite processing.

Phase 3: Consistency Assessment and Fix Recommendation. Once the final set of impacted artifacts is identified, the focus shifts from *which* artifacts are affected to *how* they are affected and *what* should be done about it. For each artifact in the Sure list, the system performs a three-step assessment:

First, it determines whether the impact causes any consistency issues. An artifact may be related to a change without requiring updates—for instance, a test file might reference changed code but still function correctly. Second, if inconsistencies are detected (such as outdated documentation or missing test cases), the system analyzes what specific updates are needed. The resolution might involve modifying the impacted file itself or updating other related artifacts. Finally, the LLM generates concrete, actionable recommendations for resolving each inconsistency, providing both an explanation of the issue and the specific changes needed.

2.3 User Interaction Model

The design prioritizes developer control and transparency throughout the analysis process. As shown in Figure 1, users interact with ArtifactSync through the VS Code extension, which provides: **Settings input:** Secure storage of API keys and preferences; **Results display:** Clear presentation of impacted files with explanations; **Apply fixes (1-click):** Direct application of recommended changes within the editor.

This design ensures that developers maintain full control over which changes are applied while benefiting from automated analysis and recommendation generation. The system presents its findings as suggestions rather than automatic modifications, allowing developers to review, understand, and selectively apply fixes based on their judgment and project requirements.

3 Implementation

This section describes how the three-phase workflow presented in Section 2 is implemented. ArtifactSync uses Python for the backend analysis engine and JavaScript for the VS Code extension frontend.

Backend (Analysis Engine). The Python backend implements the workflow described in Figure 2. In Phase 1, the system consolidates configuration from command-line arguments and environment variables, applies .gitignore rules to exclude irrelevant files, constructs the repository tree structure, retrieves commit changes via Git commands, and extracts project context from the README. The LLM communication layer supports multiple providers (OpenAI, Gemini, Anthropic) and orchestrates parallel API calls for efficient artifact processing. Phase 2 implements the hierarchical analysis by querying the LLM for uncertain items, progressively requesting additional context (file overview → full content) as needed, and recursively traversing subdirectories. Iteration limits prevent unbounded refinement loops. Phase 3 initiates parallel LLM calls to assess each impacted artifact and generate fixes when inconsistencies are detected.

Frontend (VS Code Extension). The VS Code extension implements the user interaction model shown in Figure 1. It provides secure storage for LLM provider selections and API keys, invokes the Python backend with workspace-specific parameters, displays identified impacts with natural language explanations and fix recommendations, and offers “Apply Fix” buttons that programmatically modify files through the VS Code API.

4 Evaluation

To evaluate ArtifactSync, we created 10 single-change commits for each of two open-source repositories (20 total): Crawl4AI and Unity Catalog. Each commit introduced a targeted change designed to break synchronization between artifacts (e.g., code and docs). We measured the tool’s performance at three stages: identifying the correct impact, recommending a solution, and generating the final fix. Furthermore, to simulate a common scenario where developers group numerous updates into one large commit, we combined the changes from the 10 individual scenarios into a single, massive commit for each repository. This also served as a stress test.

When analyzing the individual commits, ArtifactSync performed perfectly on all 10 Crawl4AI scenarios. For the 10 Unity Catalog

commits, it correctly identified the impact in all cases (10/10), though the recommendations and fixes were only partially correct in two instances (8/10). Our analysis traced these partial failures to the LLM returning incorrect file paths, likely due to the project's deeply nested directory structure of up to 12 levels.

The stress test with the single large commits presented a much harder challenge. Across the 20 combined changes, the tool correctly identified the impact in 80% of cases (16/20), provided a correct recommendation in 75% (15/20), and generated a correct fix in 65% (13/20). This drop in accuracy is expected, as the large commits contained multiple changes that interfered with one another.

5 Related Work

The challenge of maintaining consistency between code and other software artifacts has been a long-standing area of research. Our work builds upon traditional change impact analysis and more recent LLM-powered approaches.

Traditional and ML-based Change Impact Analysis. Early work in change impact analysis focused on static relationships, with tools like Chianti using program analysis to identify which tests might be affected by code changes [13]. Subsequent research applied machine and deep learning to predict the co-evolution of production and test code. Models like SITAR [14], CEPROT [5], and Drift [8] improved the accuracy of identifying outdated tests. While these approaches effectively prioritize maintenance tasks, they typically only detect if an update is needed, leaving developers to craft the actual fix. Our approach uses an LLM for this initial detection because its semantic reasoning is essential for the subsequent automated repair. This LLM-driven approach is also what enables our scalable, hierarchical analysis, a key aspect of our contribution.

LLM-based Artifact Maintenance. The advent of LLMs has enabled tools that automate both the identification and repair of inconsistencies, though they often focus on a single artifact type. In the testing domain, tools like ReAccept [2], UTFix [12], and Synter [7] can automatically generate corrected versions of obsolete tests. In the documentation domain, RepoAgent [10] and DocAider [6] generate and maintain repository-level documentation, while other work focuses on specific tasks like updating code comments [9]. These powerful tools, however, are typically specialized and do not address cross-artifact inconsistencies holistically.

In contrast to these works, *ArtifactSync* is designed to maintain consistency across *multiple artifact types simultaneously*. By performing a hierarchical analysis over the entire repository, our tool reasons about the relationships between production code, documentation, and tests within a unified framework. In addition to detecting out-of-sync artifacts, it also automatically proposes and applies updates, bridging the gap between detection-focused approaches and tools that target only a single type of artifact.

6 Conclusion and Future Work

Maintaining consistency among diverse software artifacts like code, tests, and documentation is a persistent challenge in software development. In this paper, we introduced *ArtifactSync*, a novel tool designed to automate this process. A core aspect of our approach is leveraging an LLM for the initial change impact analysis, not just

for the final fix generation. We find its semantic understanding is essential for enabling our scalable, hierarchical analysis and for the subsequent automated repair. Our preliminary evaluation demonstrated the tool's effectiveness across all three stages of its workflow: successfully identifying impacted artifacts, recommending appropriate solutions, and generating correct fixes. *ArtifactSync* represents a significant step towards fully automated, cross-artifact maintenance in modern software engineering.

For future work, we plan to introduce a memory to *ArtifactSync*, allowing it to learn from past analyses. Currently, our tool treats each commit independently, analyzing the repository from scratch. A memory would enable important findings, such as which artifacts are traceable to one another, to persist across commits. For example, if a pattern is identified during one repository traversal, this knowledge can be stored. By reusing these learned patterns, subsequent analyses can avoid redundant exploration, making the tool progressively faster and more token-efficient over time.

References

- [1] Paris Avgeriou, Ipek Ozkaya, Alexander Chatzigeorgiou, Marcus Ciolkowski, Neil A Ernst, Ronald J Koontz, Eltjo Poort, and Forrest Shull. 2023. Technical Debt Management: The Road Ahead for Successful Software Delivery. In *Proc. ICSE-FoSE 2023*.
- [2] Jianlei Chi, Xiaotian Wang, Yuhan Huang, Lechen Yu, Di Cui, Jianguo Sun, and Jun Sun. 2024. ReAccept: Automated Co-Evolution of Production and Test Code Based on Dynamic Validation and Large Language Models. *arXiv:2411.11033* (2024).
- [3] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. 2024. Cruxeval: A Benchmark for Code Reasoning, Understanding and Execution. *arXiv:2401.03065* (2024).
- [4] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proc. ASE 2016*.
- [5] Xing Hu, Zhuang Liu, Xin Xia, Zhongxin Liu, Tongtong Xu, and Xiaohu Yang. 2023. Identify and Update Test Cases When Production Code Changes: A Transformer-Based Approach. In *Proc. ASE 2023*.
- [6] Dias Jakupov. 2024. DocAider: Automated Documentation Maintenance for Open-Source GitHub Repositories. Microsoft Tech Community Blog. <https://techcommunity.microsoft.com/blog/educatordeveloperblog/docaider-automated-documentation-maintenance-for-open-source-github-repositories/4245588>
- [7] Jun Liu, Jiwei Yan, Yu Xie, Jun Yan, and Jian Zhang. 2024. Fix the Tests: Augmenting LLMs to Repair Test Cases with Static Collector and Neural Ranker. In *Proc. IEEE ISSRE 2024*.
- [8] Lei Liu, Sinan Wang, Yepang Liu, Jinliang Deng, and Sicen Liu. 2023. Drift: Fine-Grained Prediction of the Co-Evolution of Production and Test Code via Machine Learning. *Internetware* (2023).
- [9] Zhongxin Liu, Xin Xia, Meng Yan, and Shanning Li. 2020. Automating Just-In-Time Comment Updating. In *Proc. ASE 2020*.
- [10] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-Powered Open-Source Framework for Repository-Level Code Documentation Generation. *arXiv:2402.16667* (2024).
- [11] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. 2011. An Empirical Study of Build Maintenance Effort. In *Proc. ICSE 2011*.
- [12] Shanto Rahman, Sachit Kuhar, Berk Cirisci, Pranav Garg, Shiqi Wang, Xiaofei Ma, Anoop Deoras, and Baishakhi Ray. 2025. UTFix: Change Aware Unit Test Repairing Using LLM. *arXiv:2503.14924* (2025).
- [13] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proc. OOPSLA 2004*.
- [14] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. 2021. Understanding and Facilitating the Co-Evolution of Production and Test Code. In *Proc. IEEE SANER 2021*.
- [15] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A Large-Scale Empirical Study on Code-Comment Inconsistencies. In *Proc. ICPC 2019*.
- [16] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. 2011. Studying the Co-Evolution of Production and Test Code in Open Source and Industrial Developer Test Processes Through Repository Mining. *Empir. Softw. Eng.* 16, 3 (2011), 325–364. doi:10.1007/s10664-010-9143-7